



下载APP



43 | 生产环境：真实世界下的一个Rust项目包含哪些要素？

2021-12-08 陈天

《陈天·Rust 编程第一课》

课程介绍 >



讲述：陈天

时长 13:48 大小 12.65M

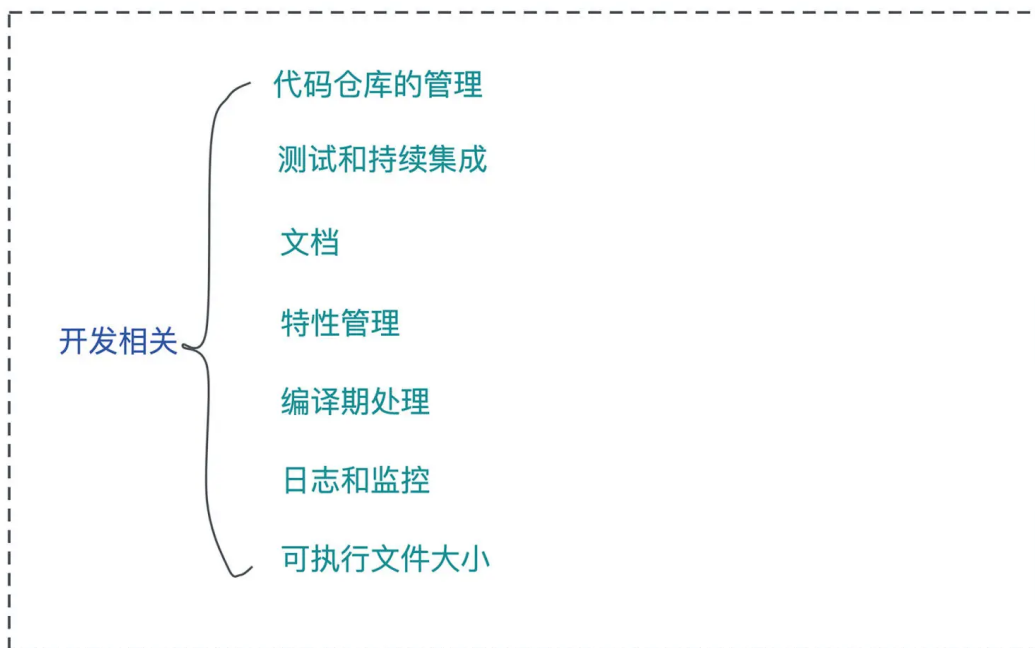


你好，我是陈天。

随着我们的实战项目 KV server 接近尾声，课程也到了收官阶段。掌握一门语言的特性，能写出应用这些特性解决一些小问题的代码，算是初窥门径，就像在游泳池里练习冲浪；真正想把语言融会贯通，还要靠大风大浪中的磨练。所以接下来的三篇文章，我们会偏重了解真实的 Rust 应用环境，看看如何用 Rust 构建复杂的软件系统。

今天，我们首先来学习真实世界下的一个 Rust 项目，应该包含哪些要素。主要介绍和开发阶段相关的内容，包括：代码仓库的管理、测试和持续集成、文档、特性管理、编译管理、日志和监控，最后会顺便介绍一下如何控制 Rust 代码编译出的可执行文件的大小。

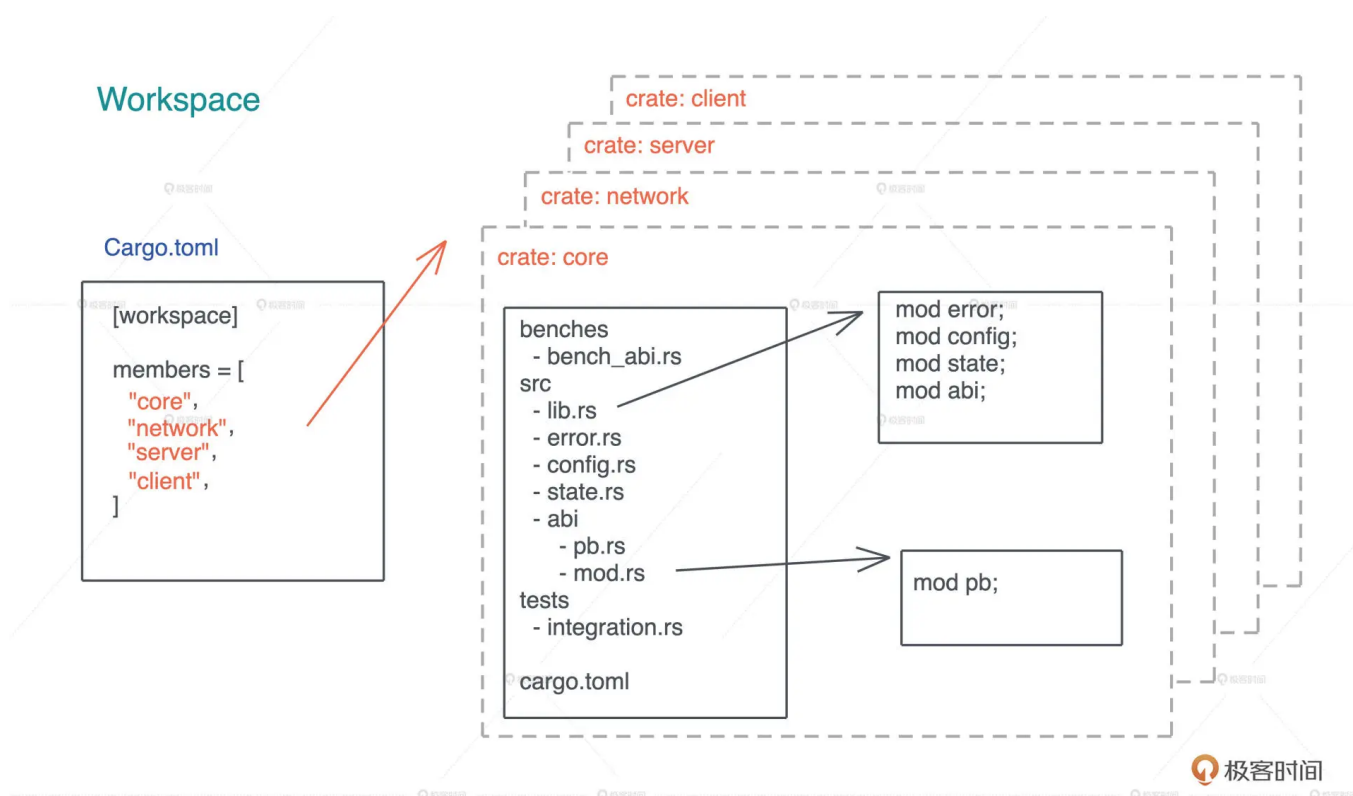




极客时间

代码仓库的管理

我们先从一个代码仓库的结构和管理入手。之前介绍过，Rust 支持 workspace，可以在一个 workspace 下放置很多 crates。不知道你有没有发现，这门课程在 GitHub 上的 repo，就把每节课的代码组织成一个个 crate，放在同一个 workspace 中。



极客时间

在构建应用程序或者服务的时候，我们要尽量把各个模块划分清楚，然后用不同的 crate 实现它们。这样，一来增量编译的效率更高（没有改动的 crate 无需重编），二来可以通过 crate 强制为模块划分边界，明确公开的接口和私有接口。

一般而言，除了代码仓库的根目录有 [🔗 README.md](#) 外，workspace 下的每个 crate 也最好要有 [🔗 README.md](#) 以及 examples，让使用者可以很清晰地理解如何使用这个 crate。如果你的项目的构建过程不是简单通过 cargo build 完成的，建议提供 Makefile 或者类似的脚本来自动化本地构建的流程。

当我们往代码仓库里提交代码时，应该要在本地走一遍基本的检查，包括代码风格检查、编译检查、静态检查，以及单元测试，这样可以最大程度保证每个提交都是完备的，没有基本错误的代码。

如果你使用 Git 来管理代码仓库，那么可以使用 pre-commit hook。一般来说，我们不必自己撰写 pre-commit hook 脚本，可以使用 [🔗 pre-commit](#) 这个工具。下面是我在 [🔗 tyrchen/geektime-rust](#) 中使用的 pre-commit 配置，供你参考：

[📄 复制代码](#)

```
1 > cat .pre-commit-config.yaml
2 fail_fast: false
3 repos:
4   - repo: <https://github.com/pre-commit/pre-commit-hooks>
5     rev: v2.3.0
6     hooks:
7       - id: check-byte-order-marker
8       - id: check-case-conflict
9       - id: check-merge-conflict
10      - id: check-symlinks
11      - id: check-yaml
12      - id: end-of-file-fixer
13      - id: mixed-line-ending
14      - id: trailing-whitespace
15   - repo: <https://github.com/psf/black>
16     rev: 19.3b0
17     hooks:
18       - id: black
19   - repo: local
20     hooks:
21       - id: cargo-fmt
22         name: cargo fmt
23         description: Format files with rustfmt.
24         entry: bash -c 'cargo fmt -- --check'
```

```
25     language: rust
26     files: \\.rs$
27     args: []
28 - id: cargo-check
29     name: cargo check
30     description: Check the package for errors.
31     entry: bash -c 'cargo check --all'
32     language: rust
33     files: \\.rs$
34     pass_filenames: false
35 - id: cargo-clippy
36     name: cargo clippy
37     description: Lint rust sources
38     entry: bash -c 'cargo clippy --all-targets --all-features --tests --be
39     language: rust
40     files: \\.rs$
41     pass_filenames: false
42 - id: cargo-test
43     name: cargo test
44     description: unit test for the project
45     entry: bash -c 'cargo test --all-features --all'
46     language: rust
47     files: \\.rs$
48     pass_filenames: false
```

你在根目录生成 `.pre-commit-config.yaml` 后，运行 `pre-commit install`，以后 `git commit` 时就会自动做这一系列的检查，保证提交代码的最基本的正确性。

除此之外，你的代码仓库在根目录下最好还声明一个 `deny.toml`，使用 [cargo-deny](#) 来确保你使用的第三方依赖没有不该出现的授权（比如不使用任何 GPL/APGL 的代码）、没有可疑的来源（比如不是来自某个 fork 的 GitHub repo 下的 commit），以及没有包含有安全漏洞的版本。

`cargo-deny` 对于生产环境下的代码非常重要，因为现代软件依赖太多，依赖树过于庞杂，靠人眼是很难审查出问题的。通过使用 `cargo-deny`，我们可以避免很多有风险的第三方库。

测试和持续集成

在课程里，我们不断地在项目中强调单元测试的重要性。单元测试除了是软件质量保证的必要手段外，还是接口设计和迭代的最佳辅助工具。

良好的架构、清晰的接口隔离，必然会让单元测试变得容易直观；而写起来别扭，甚至感觉很难撰写的单元测试，则是在警告你软件的架构或者设计出了问题：**要么是模块之间耦合性太强（状态纠缠不清），要么是接口设计得很难使用。**

在 Rust 下撰写单元测试非常直观，测试代码和模块代码放在同一个文件里，很容易阅读和互相印证。我们之前已经写过大量这类的单元测试。

不过还有一种单元测试是和文档放在一起的，doctest，如果你在学习这门课的过程中已经习惯遇到问题就去看源代码的话，会看到很多类似这样的 doctest，比如下面的

🔗 [HashMap::get](#) 方法的 doctest：

📄 复制代码

```
1  /// Returns a reference to the value corresponding to the key.
2  ///
3  /// The key may be any borrowed form of the map's key type, but
4  /// [`Hash`] and [`Eq`] on the borrowed form must match those for
5  /// the key type.
6  ///
7  /// # Examples
8  ///
9  /// ```
10 /// use std::collections::HashMap;
11 ///
12 /// let mut map = HashMap::new();
13 /// map.insert(1, "a");
14 /// assert_eq!(map.get(&1), Some(&"a"));
15 /// assert_eq!(map.get(&2), None);
16 /// ```
17 #[stable(feature = "rust1", since = "1.0.0")]
18 #[inline]
19 pub fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
20 where
21     K: Borrow<Q>,
22     Q: Hash + Eq,
23 {
24     self.base.get(k)
25 }
```

在之前的代码中，虽然我们没有明确介绍文档注释，但想必你已经知道，可以通过“`///`”来撰写数据结构、trait、方法和函数的文档注释。

这样的注释可以用 markdown 格式撰写，之后通过 “cargo doc” 编译成类似你在 docs.rs 下看到的文档。其中，markdown 里的代码就会被编译成 doctest，然后在 “cargo test” 中进行测试。

除了单元测试，我们往往还需要集成测试和性能测试。在后续 KV server 的实现过程中，我们会引入集成测试来测试服务器的基本功能，以及性能测试来测试 pub/sub 的性能。这个在遇到的时候再详细介绍。

在一个项目的早期，引入持续集成非常必要，哪怕还没有全面的测试覆盖。

如果说 pre-commit 是每个人提交代码的一道守卫，避免一些基本的错误进入到代码库，让大家在团队协作做代码审阅时，不至于还需要关注基本的代码格式；那么，**持续集成就是在团队协作过程中的一道守卫，保证添加到 PR 里或者合并到 master 下的代码，在特定的环境下，也是没有问题的。**

如果你用 GitHub 来管理代码仓库，可以使用 github workflow 来进行持续集成，比如下面是一个最基本的 Rust github workflow 的定义：

[复制代码](#)

```
1 > cat .github/workflows/build.yml
2 name: build
3
4 on:
5   push:
6     branches:
7       - master
8   pull_request:
9     branches:
10      - master
11
12 jobs:
13   build-rust:
14     strategy:
15       matrix:
16         platform: [ubuntu-latest, windows-latest]
17     runs-on: ${ matrix.platform }
18     steps:
19       - uses: actions/checkout@v2
20       - name: Cache cargo registry
21         uses: actions/cache@v1
22         with:
23           path: ~/.cargo/registry
```

```
24     key: ${{ runner.os }}-cargo-registry
25 - name: Cache cargo index
26   uses: actions/cache@v1
27   with:
28     path: ~/.cargo/git
29     key: ${{ runner.os }}-cargo-index
30 - name: Cache cargo build
31   uses: actions/cache@v1
32   with:
33     path: target
34     key: ${{ runner.os }}-cargo-build-target
35 - name: Install stable
36   uses: actions-rs/toolchain@v1
37   with:
38     profile: minimal
39     toolchain: stable
40     override: true
41 - name: Check code format
42   run: cargo fmt -- --check
43 - name: Check the package for errors
44   run: cargo check --all
45 - name: Lint rust sources
46   run: cargo clippy --all-targets --all-features --tests --benches -- -D
47 - name: Run tests
48   run: cargo test --all-features -- --test-threads=1 --nocapture
49 - name: Generate docs
50   run: cargo doc --all-features --no-deps
```

我们会处理代码格式，做基本的静态检查、单元测试和集成测试，以及生成文档。

文档

前面说了，Rust 代码的文档注释可以用 “///” 来标注。对于我们上一讲 KV server 的代码，可以运行 “cargo doc” 来生成对应的文档。

注意，在 cargo doc 时，不光你自己撰写的 crate 的文档会被生成，所有在依赖里使用到的 crate 的文档也会一并生成，所以如果你在没有网的情况下，参考某些引用了的 crate 文档，可以看本地生成的文档。下图是上一讲的 KV server 文档的截图：



Crate kv5

Version 0.1.0

See all kv5's items

Modules
Structs
Enums
Traits
Functions
Type Definitions

Crates

adler
ansi_term
anyhow
base64
bitflags
byteorder
bytes
cfg_if
chrono
core_foundation
core_foundation_sys
crc32fast
crossbeam_epoch
crossbeam_utils
dashmap
either

Crate kv5

[-][src]

Modules

command_request
value

Nested message and enum types in CommandRequest.
Nested message and enum types in Value.

Structs

Broadcaster
CommandRequest
CommandResponse
Hdel
Hexist
Hget
Hgetall
Hmdel
Hmexist
Hmget
Hmset
Hset
Kvpair
MemTable
ProstClientStream
ProstServerStream
ProstStream
Publish
Service
ServiceInner
SledDb
StorageIter
StreamResult
Subscribe
TlsClientConnector
TlsServerAcceptor
Unsubscribe
Value
YamuxCtrl

用于主题发布和订阅的数据结构
来自客户端的命令请求
服务器的响应
从 table 中删除一个 key，返回它之前的值
查看 key 是否存在
从 table 中获取一个 key，返回 value
从 table 中获取所有的 Kvpair
从 table 中删除一组 key，返回它们之前的值
查看一组 key 是否存在
从 table 中获取一组 key，返回它们的 value
往 table 中存一组 kvpair，如果 table 不存在就创建这个 table
往 table 里存一个 kvpair，如果 table 不存在就创建这个 table
返回的 kvpair
使用 DashMap 构建的 MemTable，实现了 Storage trait
处理客户端 socket 的读写
处理服务器端的某个 accept 下来的 socket 的读写
处理 KV server prost frame 的 stream
发布数据到某个主题
Service 数据结构
Service 内部数据结构
提供 Storage iterator，这样 trait 的实现者只需要 把它们的 iterator 提供给 StorageIter，然后它们保证 next() 传出的类型实现了 Into 即可
创建时之间取得 subscription id，并使用 Deref/DerefMut 使其用起来和 Stream 一致
subscribe 到某个主题，任何发布到这个主题的数据都会被收到 成功后，第一个返回的 CommandResponse，我们返回一个唯一的 subscription id
存放 TLS Client 并提供方法 connect 把底层的协议转换成 TLS
存放 TLS ServerConfig 并提供方法 accept 把底层的协议转换成 TLS
取消对某个主题的订阅
返回的值
Yamux 控制结构

大多数时候，你只需要使用 “`///`” 来撰写文档就够用了，不过如果你需要撰写 crate 级别的文档，也就是会显示在 crate 文档主页上的内容，可以在 [lib.rs](#) 或者 [main.rs](#) 的开头用 “`#![`”，比如：

```
1  ///! 这是 crate 文档
```

[复制代码](#)

如果你想强迫自己要撰写每个公共接口的文档，保持系统有良好的文档覆盖，那么可以使用 `#![deny(missing_docs)]`。这样，任何时候只要你忘记撰写文档，都会产生编译错误。如果你觉得编译错误太严格，也可以用编译报警：`#![warn(missing_docs)]`。之前我们阅读过 bytes crate 的源码，可以再回过头来看看它的 [lib.rs](#) 的开头。

在介绍测试的时候，我们提到了文档测试。

在文档中撰写样例代码并保证这个样例代码可以正常运行非常重要，因为使用者在看你的 crate 文档时，往往先会参考你的样例代码，了解接口如何使用。大部分时候，你的样例代

码该怎么写就怎么写，但是，在进行异步处理和错误处理时，需要稍微做一些额外工作。

我们来看一个文档里异步处理的例子（[🔗代码](#)）：

[📄 复制代码](#)

```
1 use std::task::Poll;
2 use futures::{prelude::*, stream::poll_fn};
3
4 /// fibonacci 算法
5 /// 示例：
6 /// ```
7 /// use futures::prelude::*;
8 /// use playground::fib; // playground crate 名字叫 playground
9 /// # futures::executor::block_on(async {
10 /// let mut st = fib(10);
11 /// assert_eq!(Some(2), st.next().await);
12 /// # });
13 /// ```
14 pub fn fib(mut n: usize) -> impl Stream<Item = i32> {
15     let mut a = 1;
16     let mut b = 1;
17     poll_fn(move |_cx| -> Poll<Option<i32>> {
18         if n == 0 {
19             return Poll::Ready(None);
20         }
21         n -= 1;
22         let c = a + b;
23         a = b;
24         b = c;
25         Poll::Ready(Some(b))
26     })
27 }
```

注意这段代码中的这两句注释：

[📄 复制代码](#)

```
1 /// # futures::executor::block_on(async {
2 /// ...
3 /// # });
```

在 /// 后出现了 #，代表这句话不会出现在示例中，但会被包括在生成的测试代码中。之所以需要 block_on，是因为调用我们的测试代码时，需要使用 await，所以需要使

步运行时来运行它。

实际上，这个的文档测试相当于：

[复制代码](#)

```
1 fn main() {
2     fn _doctest_main_xxx() {
3         use futures::prelude::*;
4         use playground::fib; // playground crate 名字叫 playground
5
6         futures::executor::block_on(async {
7             let mut st = fib(10);
8             assert_eq!(Some(2), st.next().await);
9         });
10    }
11    _doctest_main_xxx()
12 }
```

我们再来看一个文档中做错误处理的例子（[代码](#)）：

[复制代码](#)

```
1 use std::io;
2 use std::fs;
3
4 /// 写入文件
5 /// 示例：
6 /// ```
7 /// use playground::write_file;
8 /// write_file("/tmp/dummy_test", "hello world");
9 /// # Ok::<_, std::io::Error>{()}
10 /// ```
11 pub fn write_file(name: &str, contents: &str) -> Result<(), io::Error> {
12     fs::write(name, contents)
13 }
```

这个例子中，我们使用 `Result` 进行了错误处理，所以需要最后补一句 `Ok::<_, io::Error>` 来明确返回的错误类型。

如果你想了解更多有关 Rust 文档的内容，可以看 [rustdoc book](#)。

特性管理

作为一门编译型语言，Rust 支持条件编译。

通过条件编译，我们可以在同一个 crate 中支持不同的特性（feature），以满足不同的需求。比如 reqwest，它默认使用异步接口，但如果你需要同步接口，你可以使用它的“blocking”特性。

在生产环境中合理地使用特性，可以让 crate 的核心功能引入较少的依赖，而只有在启动某个特性的时候，才使用某些依赖，这样可以让最终编译出来的库或者可执行文件尽可能地小。

特性作为高级工具，并不在我们这个课程的范围内，感兴趣的话，你可以看 [cargo book](#) 深入了解如何在你的 crate 中使用特性，以及在代码撰写过程中，如何使用相应的宏来做条件编译。

编译期处理

在开发软件系统的时候，我们需要考虑哪些事情需要放在编译期处理，哪些事情放在加载期处理，哪些事情放在运行期处理。

有些事情，我们不一定要放在运行期才进行处理，可以在编译期就做一些预处理，让数据能够以更好的形式在运行期被使用。

比如在做中文繁简转换的时候，可以预先把单字对照表从文件中读取出来，处理成 `Vec<(char, char)>`，然后生成 bincode 存入到可执行文件中。我们看这个例子（[代码](#)）：

 复制代码

```
1 use std::io::{self, BufRead};
2 use std::{env, fs::File, path::Path};
3
4 fn main() {
5     // 如果 build.rs 或者繁简对照表文件变化，则重新编译
6     println!("cargo:rerun-if-changed=build.rs");
7     println!("cargo:rerun-if-changed=src/t2s.txt");
8
9     // 生成 OUT_DIR/map.bin 供 lib.rs 访问
10    let out_dir = env::var_os("OUT_DIR").unwrap();
11    let out_file = Path::new(&out_dir).join("map.bin");
12    let f = File::create(&out_file).unwrap();
```

```

13     let v = get_kv("src/t2s.txt");
14     bincode::serialize_into(f, &v).unwrap();
15 }
16
17 // 把 split 出来的 &str 转换成 char
18 fn s2c(s: &str) -> char {
19     let mut chars = s.chars();
20     let c = chars.next().unwrap();
21     assert!(chars.next().is_none());
22     assert!(c.len_utf8() == 3);
23     c
24 }
25
26 // 读取文件，把每一行繁简对照的字符串转换成 Vec<(char, char)>
27 fn get_kv(filename: &str) -> Vec<(char, char)> {
28     let f = File::open(filename).unwrap();
29     let lines = io::BufReader::new(f).lines();
30     let mut v = Vec::with_capacity(4096);
31     for line in lines {
32         let line = line.unwrap();
33         let kv: Vec<_> = line.split(' ').collect();
34         v.push((s2c(kv[0]), s2c(kv[1])));
35     }
36
37     v
38 }

```

通过这种方式，我们在编译期额外花费了一些时间，却让运行期的代码和工作大大简化（[🔗代码](#)）：

 复制代码

```

1 static MAP_DATA: &[u8] = include_bytes!(concat!(env!("OUT_DIR"), "/map.bin"));
2
3 lazy_static! {
4     /// state machine for the translation
5     static ref MAP: HashMap<char, char> = {
6         let data: Vec<(char, char)> = bincode::deserialize(MAP_DATA).unwrap();
7         data.into_iter().collect()
8     };
9     ...
10 }

```

日志和监控

我们目前撰写的项目，都还只有少量的日志。但对于生产环境下的项目来说，这远远不够。我们需要详尽的、不同级别的日志。

这样，当系统在运行过程中出现问题时，我们可以通过日志得到足够的线索，从而找到问题的源头，进而解决问题。而且在一个分布式系统下，我们往往还需要把收集到的日志集中起来，进行过滤和查询。

除了日志，我们还需要收集系统的行为数据和性能指标，来了解系统运行时的状态。

Rust 有不错的对 [🔗 prometheus](#) 的支持，比如 [🔗 rust-prometheus](#) 可以帮助你方便地收集和发送各种指标；而 `opentelemetry-rust` 更是除了支持 [🔗 prometheus](#) 外，还支持很多其他商业 / 非商业的监控工具，比如 `datadog`，比如 `jaeger`。

之后我们还会有一讲来让 KV server 更好地处理日志和监控，并且用 `jaeger` 进行性能分析，找到代码中的性能问题。

可执行文件大小

最后，我们来谈谈可执行文件的大小。

绝大多数使用场景，我们使用 `cargo build --release` 就够了，生成的 `release build` 可以用在生产环境下，但有些情况，比如嵌入式环境，或者用 Rust 构建提供给 Android/iOS 的包时，需要可执行文件或者库文件尽可能小，避免浪费文件系统的空间，或者网络流量。

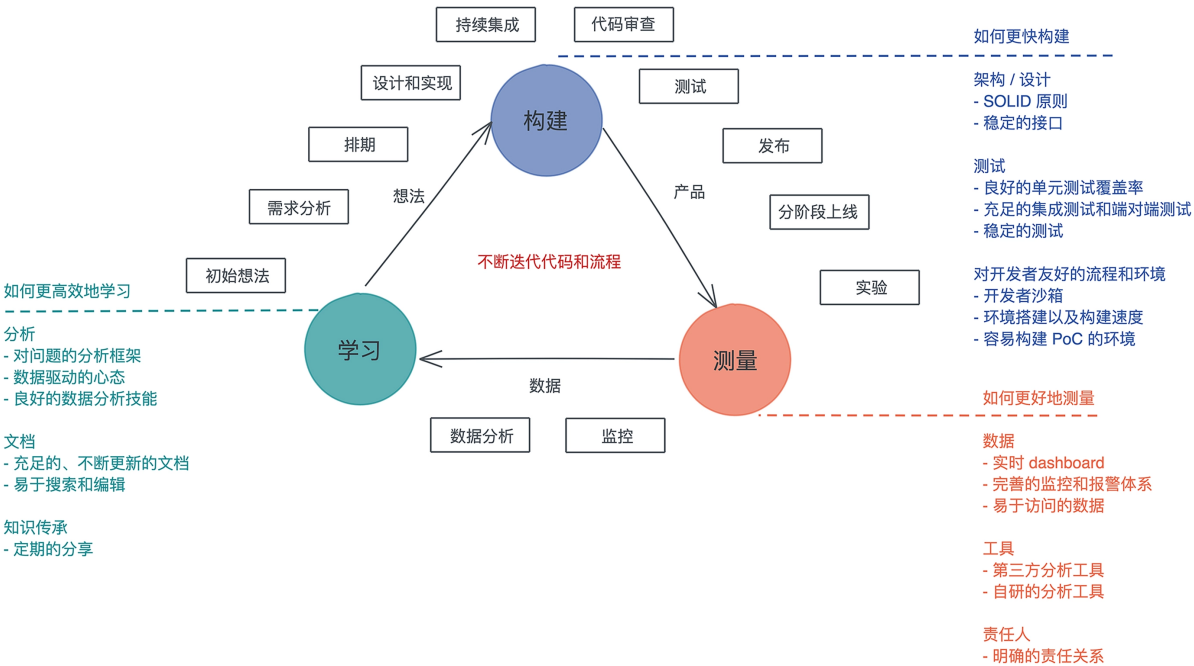
此时，我们需要一些额外的手段来优化文件尺寸。你可以参考 [🔗 min-sized-rust](#) 提供的方法进行处理。

小结

今天我们蜻蜓点水讨论了，把一个 Rust 项目真正应用在生产环境下，需要考虑的诸多问题。之后会围绕着 KV server 来实践这一讲中我们聊到的内容。



做一个业余项目和做一个实际的、要在生产环境中运行的项目有很大不同。业余项目我们主要关注需求是不是得到了妥善的实现，主要关注的是构建的流程；而在实际项目中，我们除了需要关注构建，还有测量和学习的完整开发流程。



看这张图，一个项目的整体开发流程相信是你所熟悉，包括初始想法、需求分析、排期、设计和实现、持续集成、代码审查、测试、发布、分阶段上线、实验、监控、数据分析等

部分，我把它贯穿到精益创业（Lean Startup）“构建 - 测量 - 学习”（Build - Measure - Learn）的三个环节中。

今天介绍的代码仓库的管理、测试和持续集成、文档、日志和监控，和这个流程中的很多环节都有关系，你可以对照着自己公司的开发流程，想一想如何在这些流程中更好地使用 Rust。

思考题

在上面完整开发流程图中，今天只涉及了主要的部分。你可以结合自己现有工作的流程，思考一下如果把 Rust 引入到你的工作中，哪些流程能够很好地适配，哪些流程还需要额外的工作？

欢迎在留言区分享你的思考，感谢你的收听，如果觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。你完成了 Rust 学习的第 43 次打卡啦，我们下节课见。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 42 | 阶段实操（6）：构建一个简单的KV server-如何做大的重构？

下一篇 44 | 数据处理：应用程序和数据如何打交道？

小争哥新书

数据结构与算法之美

图书+专栏，双管齐下，拿下算法

打包价 **¥159** 原价¥319

仅限 300 套



精选留言 (3)

写留言



交流会

2021-12-08

没想到rust几行代码的小项目会占这么大空间，哈哈

2.4G ./queryer

797M ./scrape_url

1.7G ./httpie

1.9G ./thumbor

展开



1



Rayjun

2021-12-08

很赞，不仅仅是 rust 项目可以参考这个流程，对于其他语言的项目，同样适用



Aaron

2021-12-08

弱弱地问一句，"加餐|开悟之坡 业界都在用rust干什么"这节课木有了么？

编辑回复: 有的，只是这是邀请的场外大咖来加餐，所以上线时间不是很稳定。不过放心，会有
的。



共 2 条评论 >

